第十三讲：用Verilog-A进行行为级建模与仿真
Lecture 13: Behavior Modeling and Simulation with Verilog-A

Zuochang Ye

2011/12/21

Outline

- Basic of Verilog-A (Verilog-AMS)
- Modeling Analog Components
- Modeling PLLs

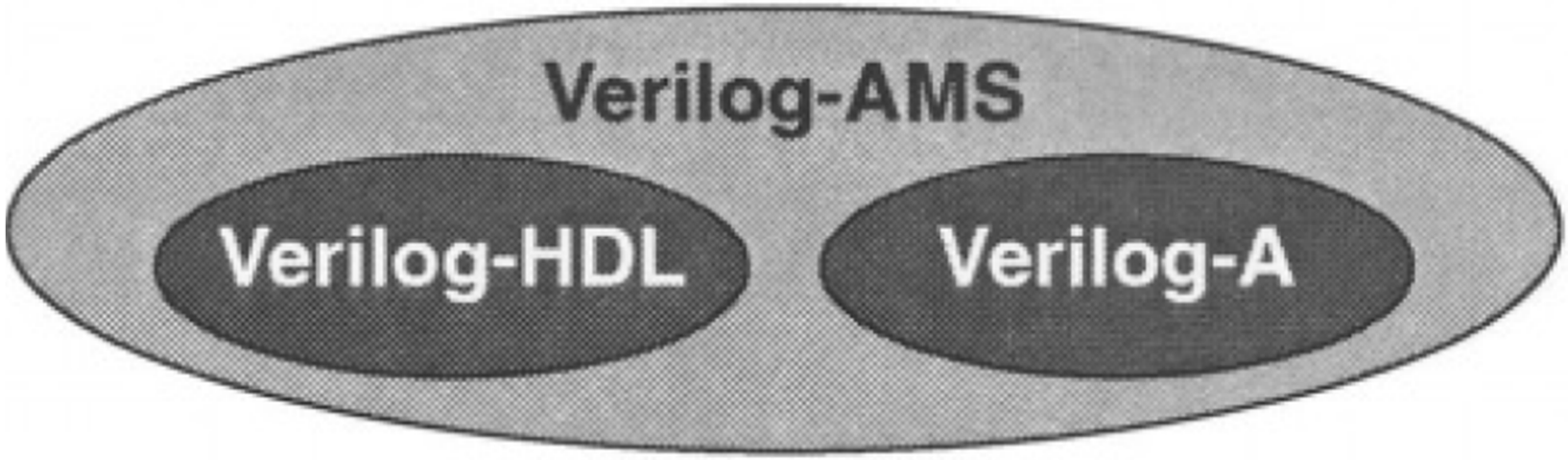
Motivation: Simulating Phase Noise of PLL

- In many circumstances, An RF simulator, such as SpectreRF, can be directly applied to predict the noise performance of a PLL.
- To make this possible, the PLL must at a minimum have a periodic steady state solution.
- To perform a noise analysis, SpectreRF must first compute the steady-state solution of the circuit with its periodic steady state (PSS) analysis.
- In practice, it is not always feasible
  - The circuit may not have a periodic solution.
  - Direct simulation is too expensive.

Introduction

- Verilog-AMS: analog and mixed signal extensions to Verilog
- An earlier language standard was called Verilog-A (Verilog with Analog extensions)
  - Verilog-A is a subset of Verilog-AMS
- Important extensions of *Verilog*-AMS over Verilog-A
  - Both digital and analog signals can be included in same module
  - User-defined conversions modules are automatically inserted in netlist if analog signal connected to digital signal or vice-versa
  - More freedom in accessing digital/analog signals within a module
- Much of the same terminology used in VHDL-AMS

The Relationship

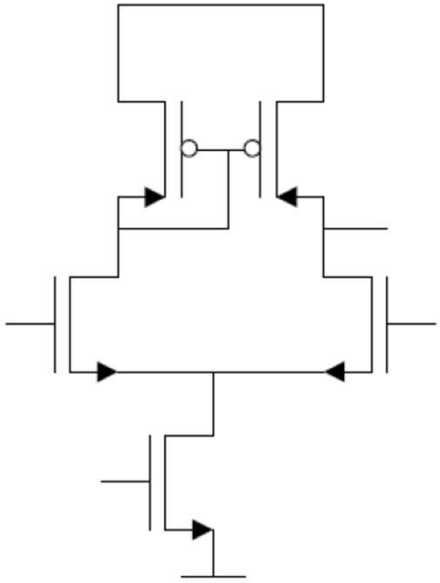


Application of Verilog-AMS

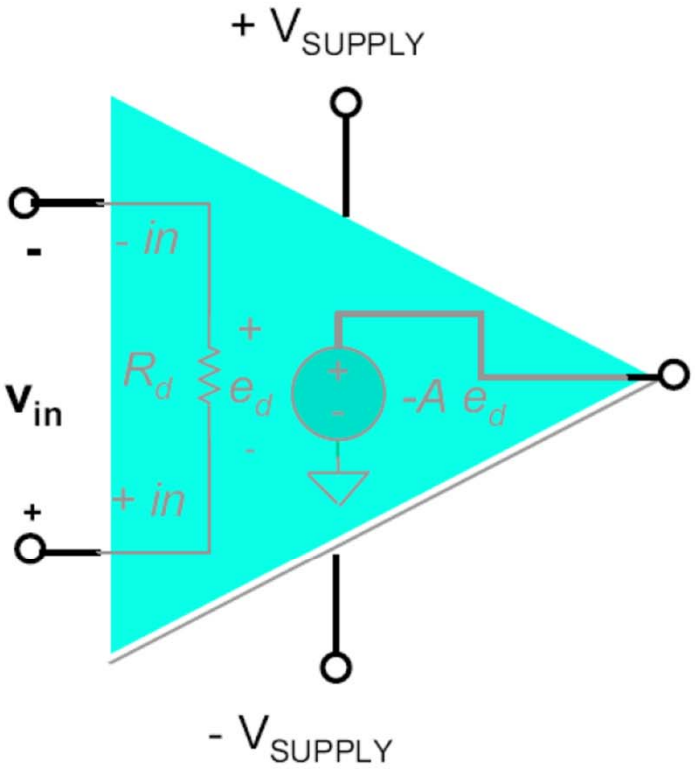
- Model components,
- Create test benches,
- Accelerate simulation,
- Verify mixed-signal systems, and
- Support the top-down design process.

Description for Mixed-Signal Systems

Circuit level



Macro Model



Analog Behavioral

```
module opamp (vout, vin_p, vin_n );
 inout  vin_p , vin_n;
 output vout;
 electrical vin_p, vin_n, vout;


analog begin

   I(vin_p)  <+ 0.0;
   I(vin_n)  <+ 0.0;
   V(vout) <+ V(vin_p, vin_n ) *Av;
 end
endmodule
```

Compromising between Accuracy and Complexity

Mixed-Signal Design

- When designing mixed-signal circuits Verilog-AMS is very useful as it allows both digital and analog circuits to be described in a way that is most suitable for each type of circuit.

- With digital circuits, either gate- or behavioral-level Verilog-HDL is used.

- With analog circuits, either transistor- or behavioral-level Verilog-A/MS is used.

Verilog-A at a Glance

```
module V_integrator(in,out);

input in;
output out;
voltage in,out;

// integration coefficient
parameter real ki=1.0 exclude 0;
parameter real dcval = 0;

real k1;
initial k1 = 1/ki;

analog

   V(out) <+ k1*idt(V(in),dcval);

endmodule
```

Port direction (*input,output,inout*)

Port *discipline*

Module parameters, can specify initial values, other limits.

Local variable

Executed at simulator startup

Behavior specified in analog block

Voltage assignment

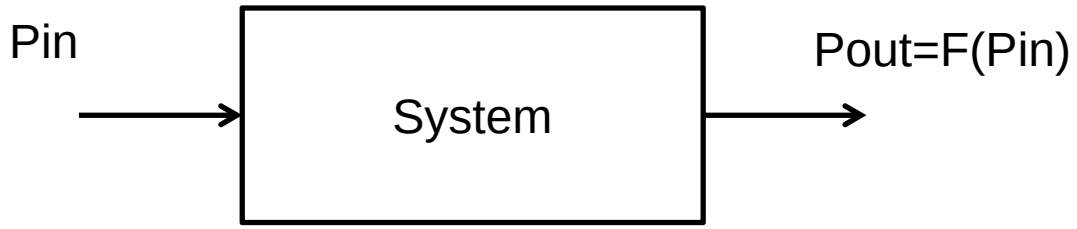
Branch assignment

Conservative System v.s. Signal-Flow System

- A *conservative* system obeys Kirchoff's laws
  - Nodes have both potential and flow
- A *signal-flow* system has only flow or potential associated with a node
  - Verilog-A supports modeling of signal-flow systems.
  - Verilog-A supports mixing of conservative and signal-flow nodes
- Physical systems are conservative systems
- Abstract systems can use a signal-flow graph model
- View potential as *across* a component (voltage, temperature, velocity)
- View flow as *through* a component (current, force, heat flow rate)

Signal-Flow System

- A discipline may specify two nature bindings, **potential** and **flow**, or it may specify only a single binding, **potential**.
- Disciplines with two natures are know as *conservative disciplines* because nodes which are bound to them exhibit Kirchhoff's Flow Law
- A discipline with only a potential nature is known as a *signal flow discipline*.

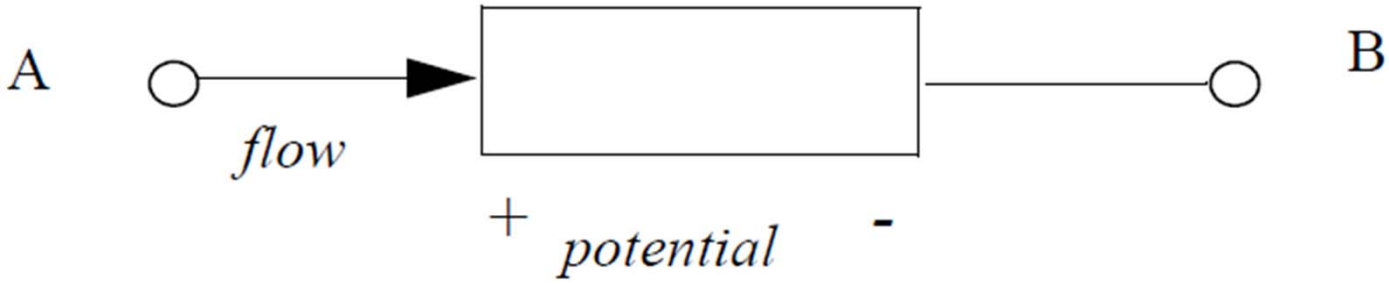


Example of a Signal-Flow System

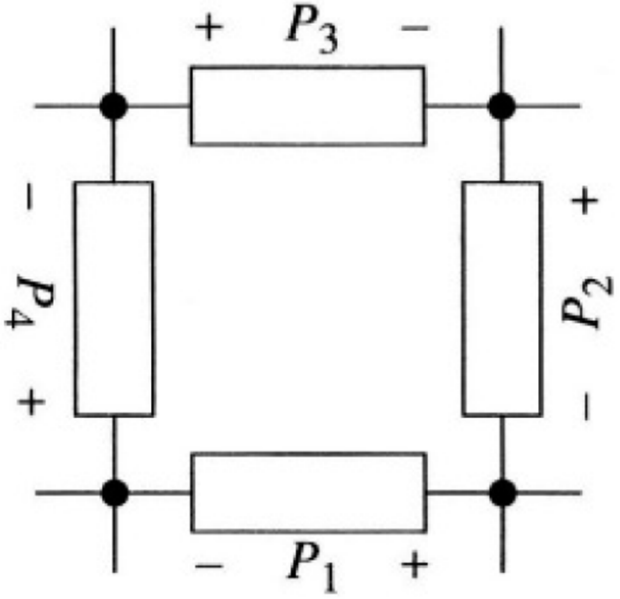
```
module shiftPlus5(in, out);
input in;
output out;
voltage in, out;  //voltage is a signal flow
                  //discipline compatible with
                  //electrical, but having a
                  //potential nature only
analog begin
   V(out) <+ 5.0 + V(in);
end
endmodule
```

Conservative systems

- An important characteristic of conservative systems is that there are two values associated with every node
  - Potential
  - Flow

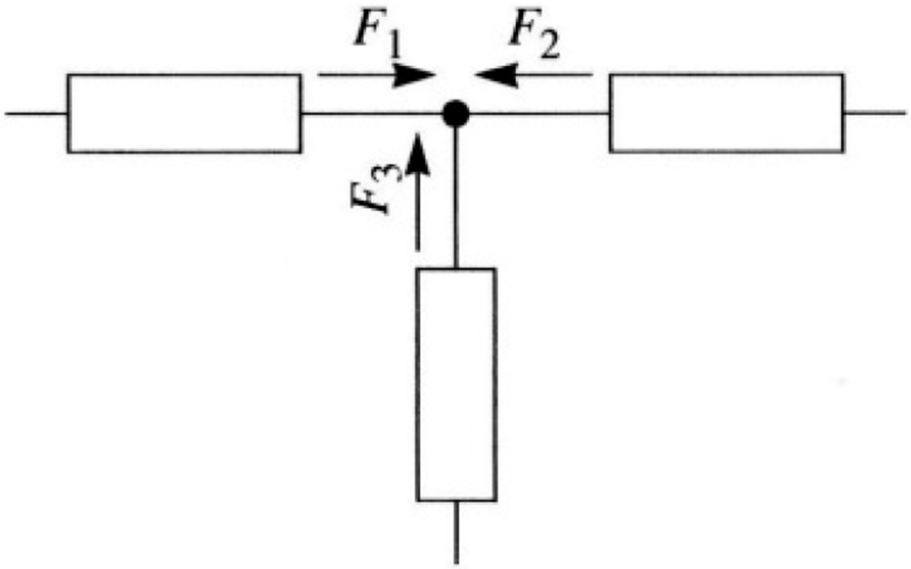


Kirchhoff's Laws



$$P_1 + P_2 + P_3 + P_4 = 0$$

Kirchhoff's Potential Law

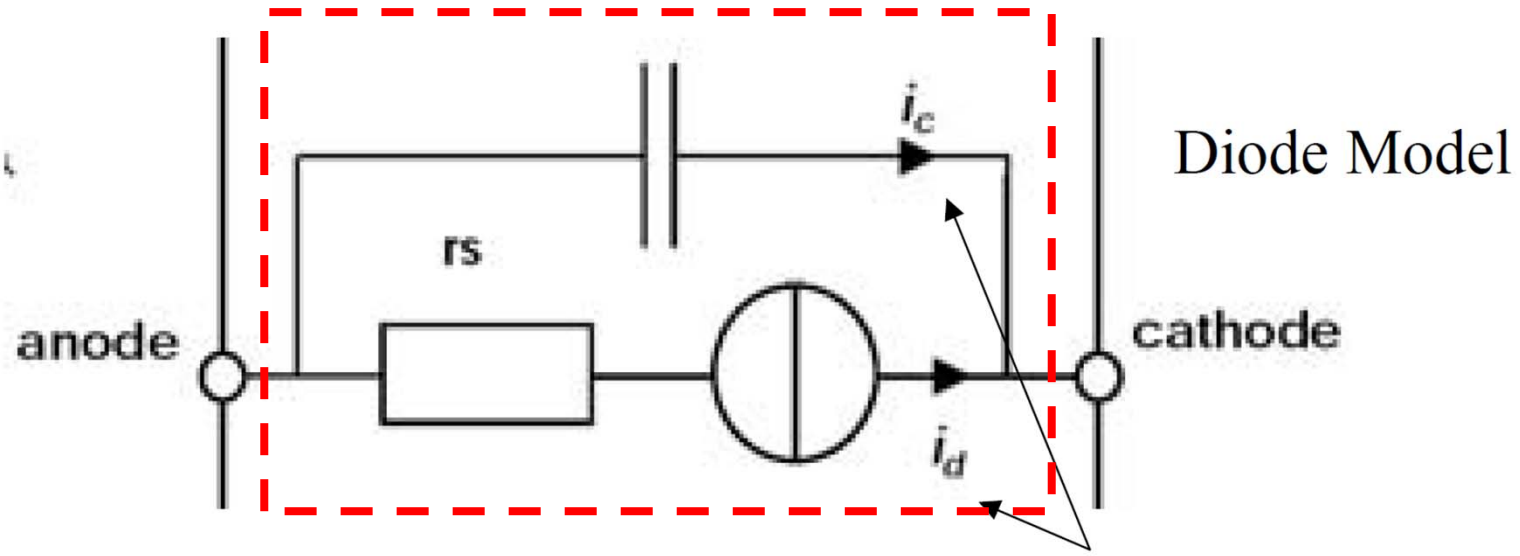


$$F_1 + F_2 + F_3 = 0$$

Kirchhoff's Flow Law

Branches

- Branch is a path between nodes
- Can only be declared within a module
- Currents summed at branch node



Define two different branch currents

Branches in Diode Model

```
module diode (a, c) ;
electrical a, c ;
branch (a, c) diode, cap ;
branch (a,a) anode;
parameter real rs = 0, is=1e-14, tf=0, cjo=0, imax=1, phi=0.7 ;

analog begin

I(diode) <+ is*($limexp((V(diode)-rs*I(anode)/$vt) - 1) ;

I(cap) <+ ddt(tf*I(diode) - 2 * cjo *
          sqrt(phi * (phi * V(cap)))) ;

if (I(anode) > imax) // Checks current through port
   $strobe( "Warning: diode is melting!" ) ;

end

endmodule
```

Special branch (port branch), used to monitor current through port

Branch currents summed

Thermal voltage, language builtin

Assignments

- Procedural assignments, used to modify integer, reals
  - `sum = a + b`
- Branch contribution statement
  - `V(n1,n2) <+ expr1;`
- Multiple branch assignments can be applied to same node

```
V(n1, n2) <+ expr1;
V(n1, n2) <+ expr2;
```

is equivalent to:

```
V(n1,n2) <+ expr1 + expr2;
```

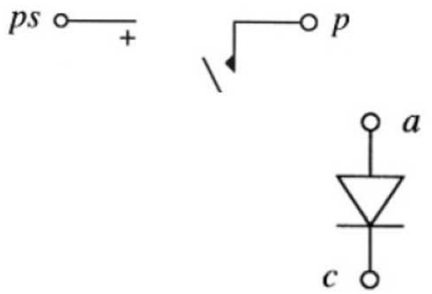
More on branch assignments

- Simulation of a branch assignment
  - Simulator evaluates right hand expression
  - Simulator adds the value of the right hand expression to any Previously retained value for the node (a summation)
  - At end of simulation cycle, summed value assigned to source branch

- Any branch, either explicit or implicit, is a ***source branch*** if either the potential or the flow of that branch is assigned a value by a contribution statement anywhere in the module.
  - It is a potential source if the branch potential is specified and is a flow source if the branch flow is specified.
  - A branch cannot simultaneously be both a potential and a flow source, although it can switch between them.
  - If assigning a flow quantity, and previously assigned value was a potential, then potential value is discarded (and vice versa)

Relay as a Switch Branch

```
// Ideal relay
`include "disciplines.vams"

// Ideal diode
`include "disciplines.vams"

module diode (a, c);
    inout a, c;
    electrical a, c;

    analog begin
        @(cross((V(a,c) + I(a,c)), 0))
            ;
        if ((V(a,c) + I(a,c)) > 0)
            V(a,c) <+ 0;
        else
            I(a,c) <+ 0;
    end
endmodule
```

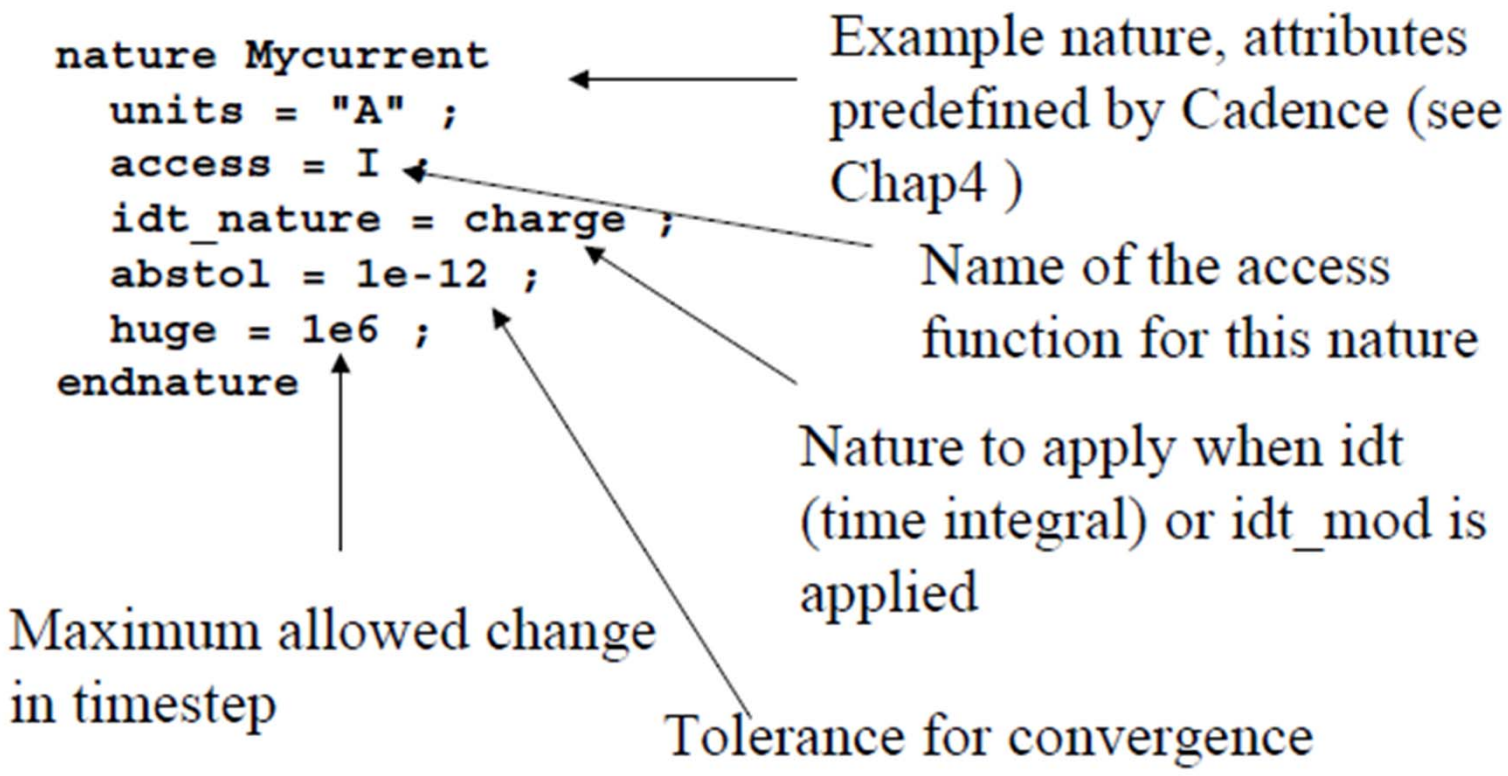


Attribute, Nature and Discipline

- Attributes define the value of certain quantities which characterize the nature.
- A *nature* is a collection of attributes.
- A *discipline* description consists of specifying a **domain** type and binding any *natures* to **potential** or **flow**.

Natures and Disciplines

- A nature is a collection of attributes
  - Attributes characterize quantities solved for during simulation



Some Pre-defined Natures

```
nature Current
 units      = "A";
 access     = I;
 idt_nature = Charge;
endnature
```

```
nature Voltage
 units      = "V";
 access     = V;
 idt_nature = Flux;
endnature
```

```
nature Charge
 units      = "coul";
 access     = Q;
 ddt_nature = Current;
endnature"
```

```
nature Flux
 units      = "Wb";
 access     = Phi;
 ddt_nature = Voltage;
endnature"
```

Defined in “discipline.h” include file

Disciplines

- Disciplines used to bind natures with potential and flow

```
discipline voltage
  potential Voltage;
enddiscipline

discipline current
 potential  Current;
enddiscipline

discipline electrical
  potential  Voltage;
  flow       Current;
enddiscipline
```

Discipline with single nature called *signal-flow* discipline

Discipline with multiple natures called *conservative* discipline. Nature bound to potential must be different from nature bound to flow.

Analog Operations

- Built-in functions that operate on more than just the current
- value of their arguments – they maintain internal state
  - Limited Exponential function ($limexp)
  - Time derivative operator (ddt)
  - Time Integral operator (idt)
  - Circular integrator operator (idtmod)
  - Delay operator (delay)
  - Transition filter (transition)
  - Slew filter (slew)
  - Laplace transform filters (laplace_zp, laplace_zd, laplace_np, laplace_nd)
  - Z-transform filters (zi_zp, zi_zd, zi_np, zi_nd)

Time derivative

Operator	Comments
ddt(*expr*)	Returns $\frac{d}{dt}x(t)$, the time-derivative of *x, where x is expr.*
ddt(*expr, abstol*)	Same as above, except absolute tolerance is specified explicitly.
ddt(*expr, nature*)	Same as above, except nature is specified explicitly.

Example of Time Derivative

```
module opamp(out, pin, nin);
    output out;
    input pin, nin;
    voltage out, pin, nin;
    analog
        V(out) <+ idt(V(pin,nin));
endmodule
```

Time Integral

Operator	Comments
idt(*expr*)	Returns $\int_{t_0}^{t} x(\tau)d\tau + c$, where $x(\tau)$ is the value of *expr* at time τ, t_0 is the start time of the simulation, t is the current time, and c is the initial starting point as determined by the simulator and is generally the DC value (the value that makes expr equal to zero).
idt(*expr,ic*)	Returns $\int_{t_0}^{t} x(\tau)d\tau + c$, where in this case c is the value of *ic* at t_0.
idt(*expr,ic,assert*)	Returns $\int_{t_a}^{t} x(\tau)d\tau + c$, where c is the value of *ic* at t_a, which is the time when *assert* was last nonzero or t_0 if *assert* was never nonzero.
idt(*expr,ic,assert,abstol*)	Same as above, except the absolute tolerance used to control the error in the numerical integration process is specified explicitly with *abstol*.
idt(*expr,ic,assert,nature*)	Same as above, except the absolute tolerance used to control the error in the numerical integration process is take from the specified nature.

Example of Time Integral

```
module opamp(out, pin, nin);
    output out;
    input pin, nin;
    voltage out, pin, nin;
    analog
        V(out) <+ idt(V(pin,nin));
endmodule
```

- In this case the initial condition for the integrator is found by the simulator, generally the DC operating point is used.
- Forcing the output of the integration operator to be a particular value at start of the simulation using something like

```
V(out) <+ idt(V(pin,nin), 0);
```

Expression Derivative operator

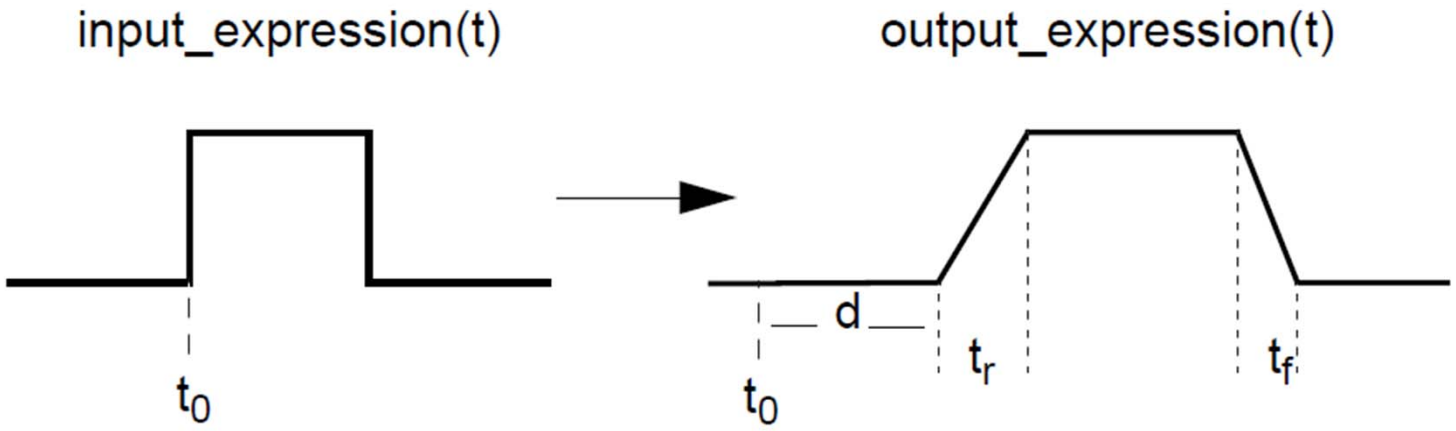
ddx (expr , unknown_quantity)

- *expr* is the expression for which the symbolic derivative needs to be calculated.
- *unknown_quantity* is the branch probe (voltage or current probe) with respect to which the derivative of the expression needs to be computed.

```
module diode(a,c);
   inout a, c;
   electrical a, c;
   parameter real IS = 1.0e-14;
   real idio;
   (*desc="small-signal conductance"*)
   real gdio;
   analog begin
      idio = IS * (limexp(V(a,c)/$vt) - 1);
      gdio = ddx(idio, V(a));
      I(a,c) <+ idio;
   end
endmodule
```

Transistion

- *transition* smooths out piece-wise constant waveforms. The transition filter is used to imitate transitions and delays on digital signals.
- This function provides controlled transitions between discrete signal levels by setting the rise time and fall time of signal transitions. *transition* stretches instantaneous changes in signals over a finite amount of time, as shown below, and can delay the transitions.



Analysis dependent functions

- The **analysis()** function takes one or more string arguments and returns one (1) if any argument matches the current analysis type. Otherwise it returns zero (0).

Name	Analysis description
"ac"	.AC analysis
"dc"	.OP or .DC analysis (single point or dc sweep analysis)
"noise"	.NOISE analysis
"tran"	.TRAN analysis
"ic"	The initial-condition analysis which precedes a transient analysis.

```
if (analysis("ic"))
   V(cap) <+ initial_value;
else
   I(cap) <+ ddt(C*V(cap));
```

Noise

- Several functions are provided to support noise modeling during **small-signal analyses**.
- To model large-signal noise during transient analyses, use the **$random()** or **$arandom()** system tasks.
- **NOTICE:** build-in noise are only available for small signal analysis.

White Noise

- White noise processes are those whose current value is completely uncorrelated with any previous or future values. This implies their spectral density does not depend on frequency.

```
white_noise ( pwr [ , name ] )
```

```
I(a,b) <+ V(a,b)/R +
    white_noise(4 * 'P_K * $temperature/R, "thermal");
```

Flicker Noise

- The **flicker_noise()** function models flicker noise. The general form is:

```
flicker_noise ( pwr , exp [ , name ] )
```

which generates pink noise with a power of *pwr* at 1Hz which varies in proportion to $1/f^{exp}$.

Noise Table

- The **noise_table()** function interpolates a set of values to model a process where the spectral density of the noise varies as a piecewise linear function of frequency.
- The general form is:

```
noise_table ( input [ , name ] )
```

```
# noise_table_input.tbl
# Example of input file format for noise_table
#
#      freq          pwr
       1.0e0     1.657580e-23
       1.0e1     3.315160e-23
       1.0e2     6.636320e-23
       1.0e3     1.326064e-22
       1.0e4     2.652128e-22
       1.0e5     5.304256e-22
       1.0e6     1.060851e-21
```

Noise Model for Diode

- The noise of a junction diode could be modelled as shown in the following example.

```
I(a,c) <+ is*(exp(V(a,c) / (n * $vt)) - 1)
    + white_noise(2*`P_Q*I(<a>))
    + flicker_noise(kf*pow(abs(I(<a>)), af), ef);
```

Analog Events

- An event is an occurrence of a particular change in the state of the circuit. They are detected by setting up a statement that looks for the desired change.
- When the event occurs, an action is taken.

$$@\,(event\text{-}expression)$$
$$action;$$

- The analog behavior of a component can be controlled using analog events.

Analog Events

- The analog behavior of a component can be controlled using analog events.

```
module bitErrorRate (in, ref) ;
   input in, ref ;
   electrical in, ref ;
   parameter real period=1, thresh=0.5 ;
   integer bits, errors ;

   analog begin
      @(initial_step) begin
            bits = 0 ;
            errors = 0 ;
      end
      @(timer(0, period)) begin
            if ((V(in) > thresh) != (V(ref) > thresh))
               errors = errors + 1 ;
            bits = bits + 1 ;
      end
      @(final_step)
        $strobe("bit error rate = %f%%", 100.0 * errors / bits ) ;
   end
endmodule
```

Global Events

- **initial_step** and **final_step** generate global events on the first and the last point in an analysis respectively.

Examples:

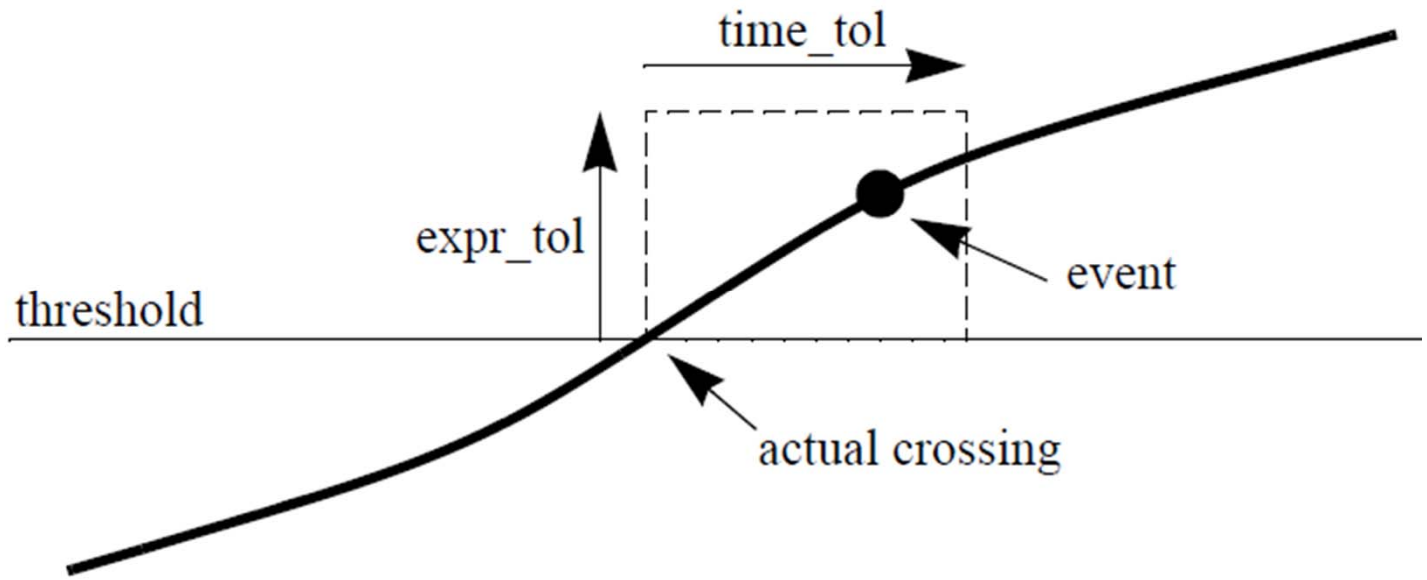
```
@(initial_step("ac", "dc")) // active for dc and ac only
@(initial_step("tran"))       // active for transient only
```

cross() Function

- The *cross function* is used for generating a monitored analog event to detect threshold crossings in analog signals.

```
module sh (in, out, smpl) ;
    output out ;
    input in, smpl ;
    electrical in, out, smpl ;
    real state ;

    analog begin
        @(cross(V(smpl) - 2.5, +1))
            state = V(in) ;
        V(out) <+ transition(state, 0, 10n) ;
    end
endmodule
```

Timing of event relative to threshold crossing



- The event shall occur after the threshold crossing, and while the signal remains in the box defined by actual crossing and *expr_tol* and *time_tol*.

timer() Function

- The *timer function* is used to generate analog event to detect specific points in time.

```
module bitStream (out) ;
   output out ;
   electrical out ;
   parameter period = 1.0 ;
   integer x ;

   analog begin
      @(timer(0, period))
            x = $random + 0.5 ;
      V(out) <+ transition( x, 0.0, period/100.0 ) ;
   end
endmodule
```

Event OR operator

- The “OR-ing” of events indicates the occurrence of any one of the events specified shall trigger the execution of the procedural statement following the event.

For example,

```
analog begin
    @(initial_step or cross(V(smpl)-2.5,+1))
        V(out) <+ 0 ;
end
```

Here, initial_step is a global event and cross() returns a monitored event. V(out) is set to 0 when one of the two events occur.

Modeling Circuit Components

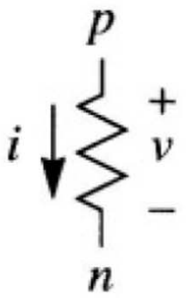
- Resistor
- Capacitor
- Inductor
- Voltage source
- Current source
- Voltage controlled voltage source

Resistor

- Equation

$$i = gv$$

```
// Linear resistor (conductance formulation)
`include "disciplines.vams"
module conductor (p, n);
    parameter real g=0; // conductance (Siemens)
    inout p, n;
    electrical p, n;

    analog
        I(p,n) <+ g * V(p,n);
endmodule
```



$v = V(p,n)$

$i = I(p,n)$

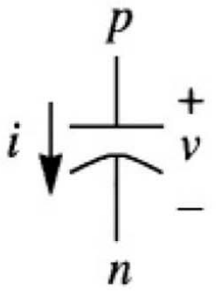
$i = gv$

Capacitor

- Equation

$$i = C\frac{d}{dt}v$$

```
// Linear capacitor
`include "disciplines.vams"
module capacitor (p, n);
    parameter real c=0; // capacitance (F)
    inout p, n;
    electrical p, n;

    analog
        I(p,n) <+ c * ddt(V(p,n));
endmodule
```



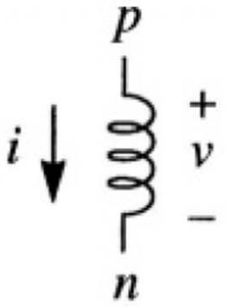
$$v = V(p,n)$$
$$i = i(p,n)$$
$$i = c\frac{dv}{dt}$$

Inductor

- Equation

$$v = L\frac{d}{dt}i$$

```
// Linear inductor
`include "disciplines.vams"
module inductor(p, n);
    parameter real l=0; // inductance (H)
    inout p, n;
    electrical p, n;

    analog
        V(p,n) <+ l * ddt(I(p,n));
    endmodule
```



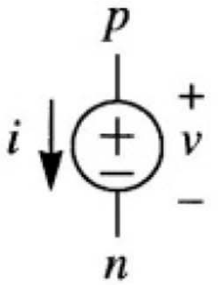
$v = V(p,n)$

$i = I(p,n)$

$v = l\frac{di}{dt}$

Voltage Source

```
// DC voltage source
`include "disciplines.vams"
module vsrc (p, n);
    parameter real dc=0;      // dc voltage (V)
    output p, n;
    electrical p, n;

    analog
        V(p,n) <+ dc;
endmodule
```



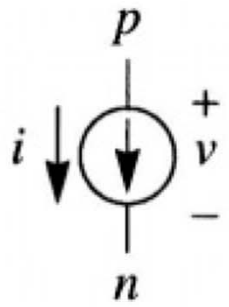
$v = V(p,n)$

$i = I(p,n)$

$v = v_{dc}$

Current Source

```
// DC current source
`include "disciplines.vams"
module isrc (p, n);
    parameter real dc=0;     // dc current (A)
    output p, n;
    electrical p, n;
    analog
        I(p,n) <+ dc;
endmodule
```



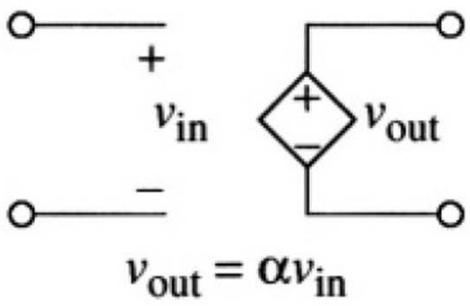
$v = V(p,n)$

$i = I(p,n)$

$i = i_{dc}$

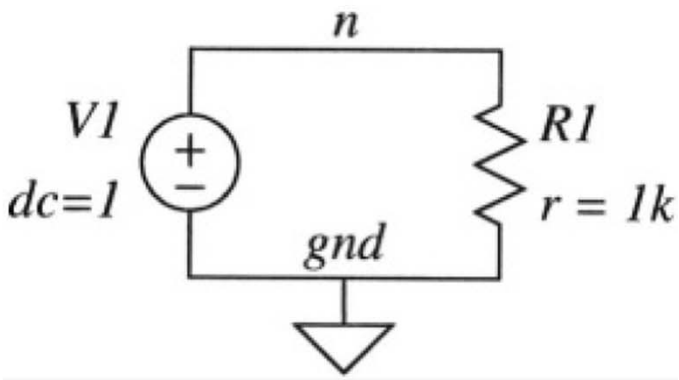
Voltage Controlled Voltage Source

```
// Voltage-controlled voltage source
`include "disciplines.vams"

module vcvs (p, n, ps, ns);
    parameter real gain=1;      // voltage gain (V/V)
    output p, n;
    input ps, ns;
    electrical p, n, ps, ns;

    analog
        V(p,n) <+ gain*V(ps,ns);
endmodule
```



Connecting Components within Verilog-A

```
//A simple circuit
`include "disciplines.vams"
`include "vsrc.vams"
`include "resistor.vams"

module smpl_ckt;
    electrical n;
    ground gnd;

    vsrc #(.dc(1)) V1(n, gnd);
    resistor #(.r(1k)) R1(n, gnd);
endmodule
```



Series RLC Circuit

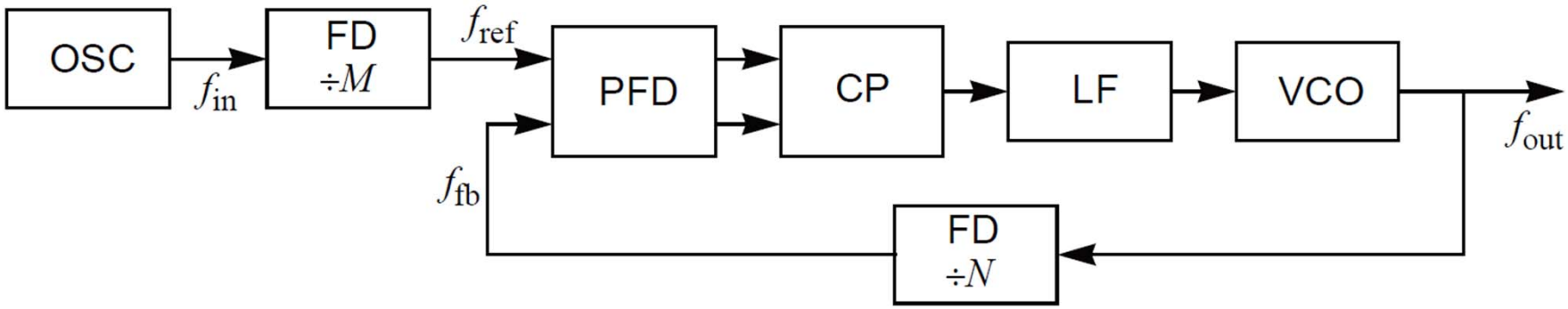
```
// Series RLC

`include "disciplines.vams"

module series_rlc (p, n);
    parameter real r=0;
    parameter real l=0;
    parameter real c=1p exclude 0;
    inout p, n;
    electrical p, n;

    analog begin
        V(p,n) <+ r*I(p,n);
        V(p,n) <+ l*ddt(I(p,n));
        V(p,n) <+ idt(I(p,n))/c;
    end
endmodule
```

Shunt RLC Circuit

```
// Shunt RLC
`include "disciplines.vams"
module shunt_rlc (p, n);
    parameter real r=1 exclude 0;
    parameter real l=1n exclude 0;
    parameter real c=0;
    inout p, n;
    electrical p, n;
    analog begin
        I(p,n) <+ V(p,n)/r;
        I(p,n) <+ c*ddt(V(p,n));
        I(p,n) <+ idt(V(p,n))/l;
    end
endmodule
```

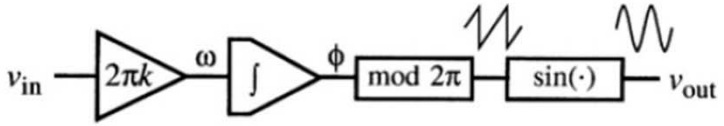
Modeling PLL

- VCO
- Jitter



Modeling VCO

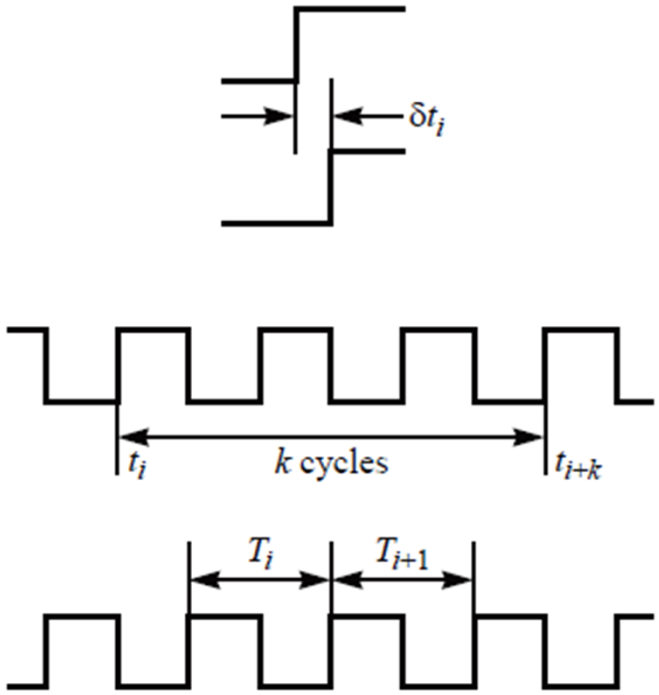
```
// Voltage-controlled oscillator
`include "disciplines.vams"
`include "constants.vams"
```



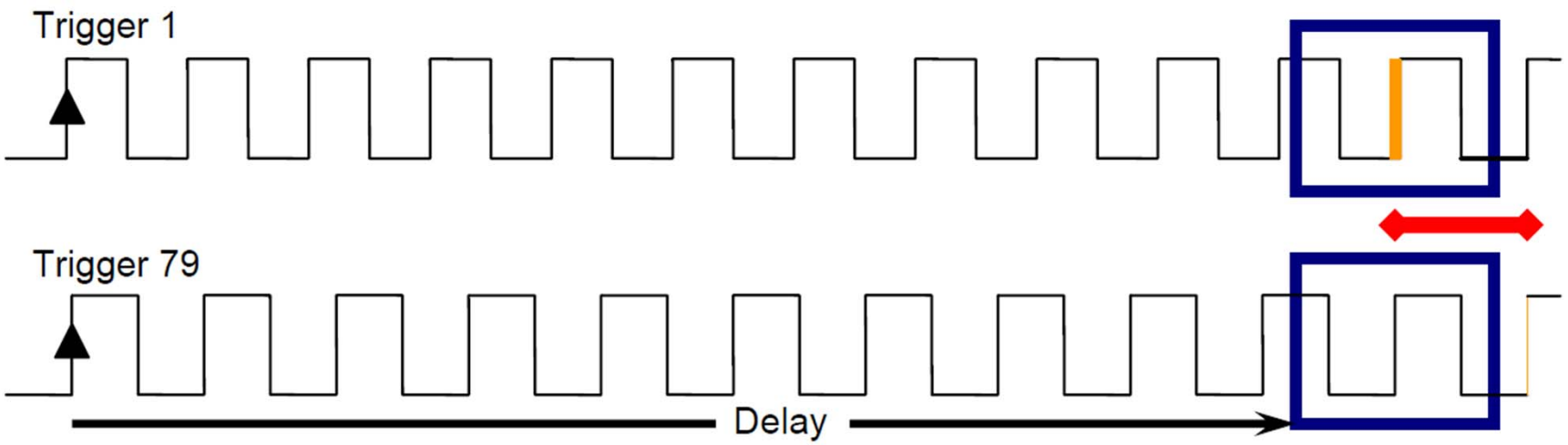

```
module vco (out, in);
    parameter real Vmin=0;                          //minimum input voltage (V)
    parameter real Vmax=Vmin+1 from (Vmin:inf);     //maximum input voltage (V)
    parameter real Fmin=1 from (0:inf);             //minimum output freq (Hz)
    parameter real Fmax=2*Fmin from (Fmin:inf);     //maximum output freq (Hz)
    parameter real ampl=1;                          //output amplitude (V)
    input in; output out;
    voltage out, in;
    real freq, phase;

    analog begin
        //compute the freq from the input voltage
        freq = (V(in) – Vmin)*(Fmax – Fmin) / (Vmax – Vmin) + Fmin;

        //bound the frequency (this is optional)
        if (freq > Fmax) freq = Fmax;
        if (freq < Fmin) freq = Fmin;

        //phase is the integral of the freq modulo 2π
        phase = 2*`M_PI*idtmod(freq, 0.0, 1.0, –0.5);

        //generate the output
        V(out) <+ sin(phase);

        //bound the time step
        $bound_step(0.1/ freq);
    end
endmodule
```

Jitter Metrics

- Define {*ti*} as the sequence of times for positive-going threshold crossings, henceforth referred to as *transitions*, that occur in *v*n. Various jitter metrics characterize the statistics of this sequence.
- Edge-to-edge jitter assumes an input signal, and so is only defined for **driven systems**.
- Cycle-to-cycle jitter assumes an input signal, and so is only defined for **driven systems**.



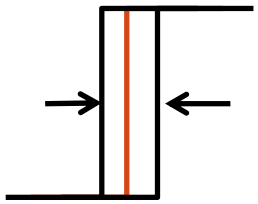
Modeling Jitters

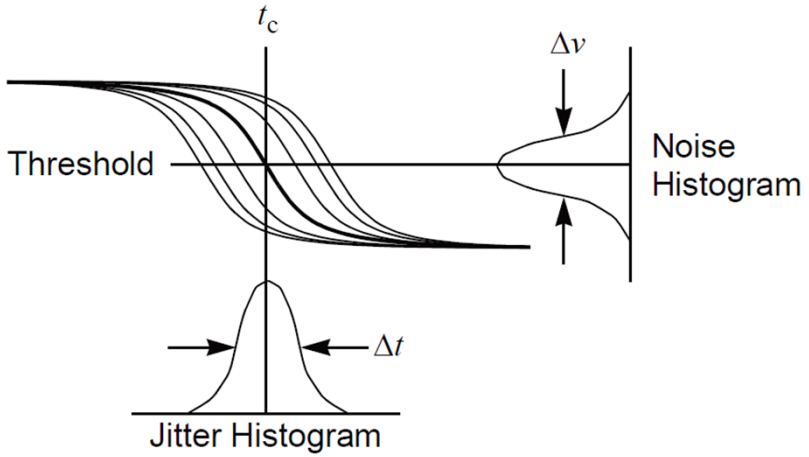


- Synchronous Jitter
- Accumulating Jitter

Synchronous Jitter

- Blocks such as the PFD, CP, and FD are driven, meaning that a transition at their output is a direct result of a transition at their input.
- The jitter exhibited by these blocks is referred to as *synchronous jitter*, it is a variation in the delay between when the input is received and the output is produced.
- The ***transition()*** function can be used to model synchronous jitter.





Modeling Synchronous Jitter in Divider

```
`include "disciplines.vams"

module divider (out, in);

input in; output out; electrical in, out;

parameter real Vlo=–1, Vhi=1;
parameter integer ratio=2 from [2:inf);
parameter integer dir=1 from [–1:1] exclude 0;  // dir=1 for positive edge trigger
                                                // dir=–1 for negative edge trigger
parameter real tt=1n from (0:inf);
parameter real td=0 from (0:inf);
parameter real jitter=0 from [0:td/5);          // edge-to-edge jitter
parameter real ttol=1p from (0:td/5);           // recommend ttol << jitter

integer count, n, seed;
real dt;

analog begin
    @(initial_step) seed = –311;

    @(cross(V(in) – (Vhi + Vlo)/2, dir, ttol)) begin
        // count input transitions
        count = count + 1;
        if (count >= ratio)
            count = 0;
        n = (2*count >= ratio);
        // add jitter
        dt = jitter*$rdist_normal(seed,0,1);
    end

    V(out) <+ transition(n ? Vhi : Vlo, td+dt, tt);
end
endmodule
```

Modeling Synchronous Jitter in PFD/CP

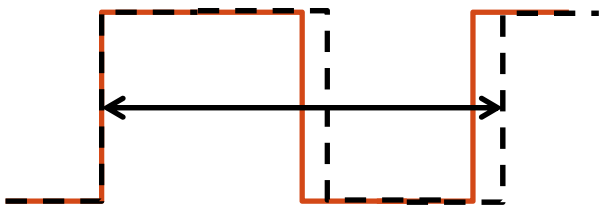
- PFD/CP can be implemented as a finite-state machine with a three-level output, –/out, 0 and +/out.
- On every transition of the VCO input in direction dir, the output is incremented. On every transition of the reference input in the direction dir, the output is decremented.
- If both the VCO and reference inputs are at the same frequency, then the average value of the output is proportional to the phase difference between the two, with the average being negative if the reference transition leads the VCO transition and positive other-wise.
- The times of the output transitions are randomly dithered by dt to model jitter. The output is modeled as an ideal current source and a finite transition time provides a simple model of the dead band in the CP.

Modeling Synchronous Jitter in PFD/CP

```
`include "disciplines.vams"

module pfd_cp (out, ref, vco);

input ref, vco; output out; electrical ref, vco, out;

parameter real Iout=100u;
parameter integer dir=1 from [–1:1] exclude 0;  // dir=1 for positive edge trigger
                                                // dir=—1 for negative edge trigger
parameter real tt=1n from (0:inf);
parameter real td=0 from (0:inf);
parameter real jitter=0 from [0:td/5);          // edge-to-edge jitter
parameter real ttol=1p from (0:td/5);           // recommend ttol << jitter

integer state, seed;
real dt;

analog begin
    @(initial_step) seed = 716;

    @(cross(V(ref), dir, ttol)) begin
        if (state > –1) state = state – 1;
        dt = jitter*$rdist_normal(seed,0,1);
    end

    @(cross(V(vco), dir, ttol)) begin
        if (state < 1) state = state + 1;
        dt = jitter*$rdist_normal(seed,0,1);
    end

    I(out) <+ transition(Iout*state, td + dt, tt);
end
endmodule
```

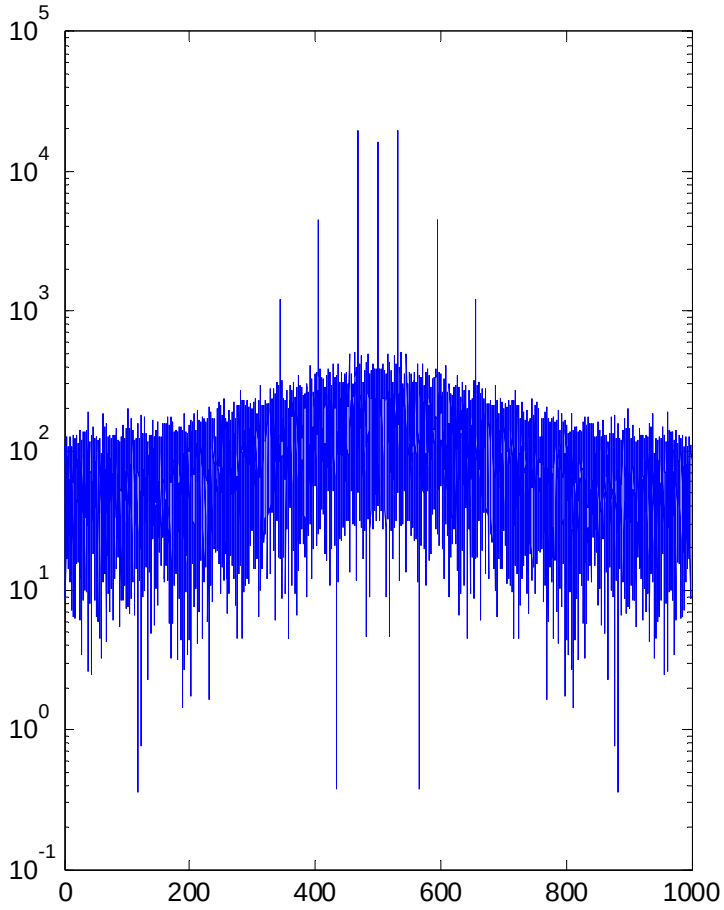
Accumulating Jitter

- Blocks such as the OSC and VCO are autonomous.
- They generate output transitions not as a result of transitions at their inputs, but rather as a result of the previous output transition.

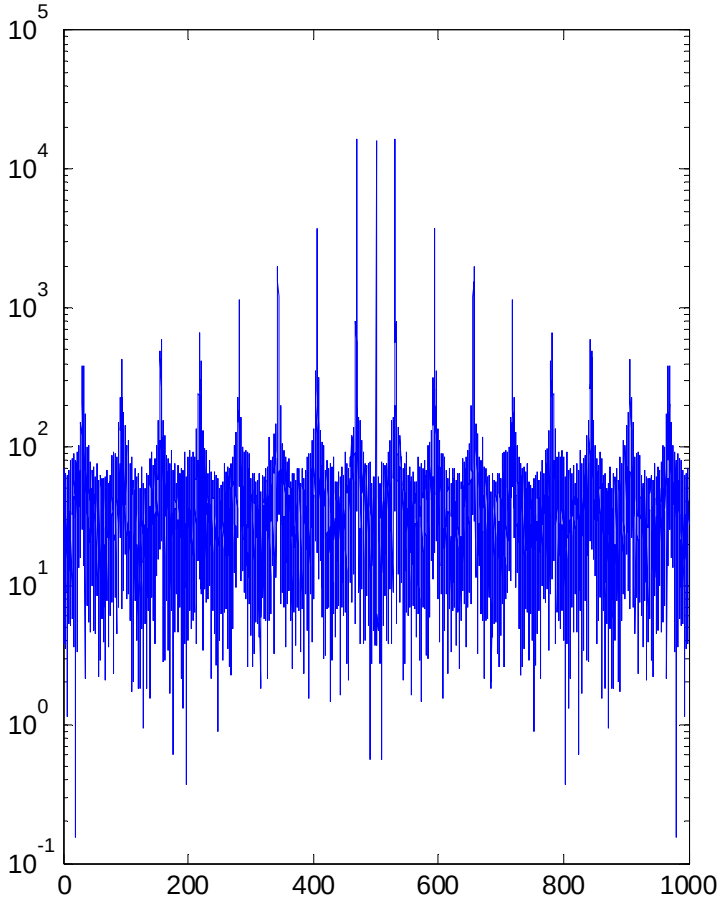


- Generally, the jitter produced by the OSC and VCO are well approximated by simple accumulating jitter if one can neglect flicker noise.
- The delay argument of the ***transition*()** function cannot be used to model accumulating jitter because of the unbounded nature of this type of jitter.
- User ***timer()*** to model accumulating jitter.

Spectrum of Synchronous and Accumulating Jitters



Synchronous



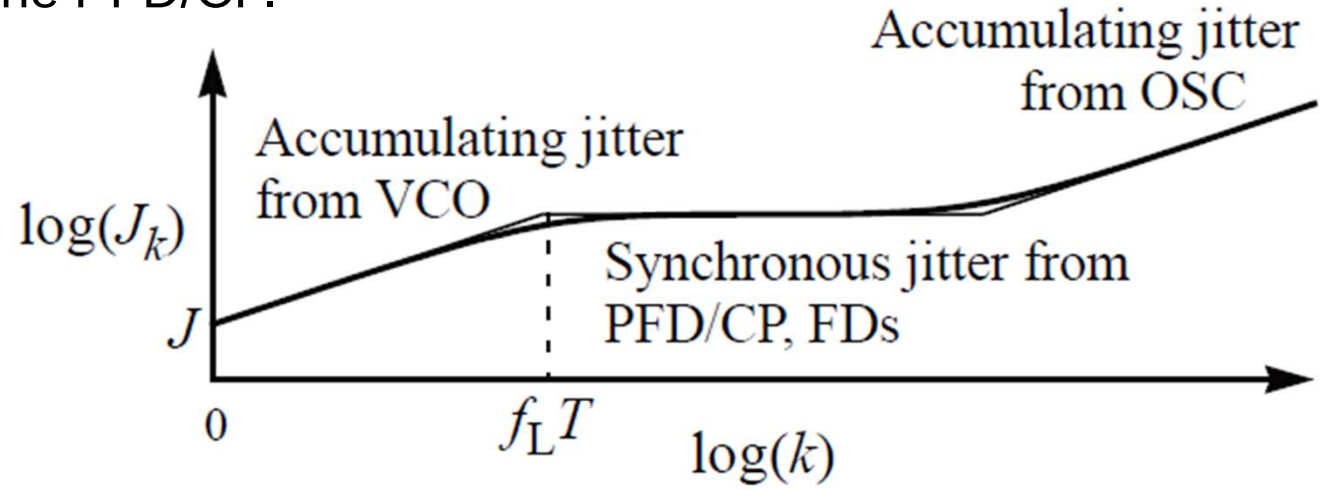
Accumulating

Modeling Accumulating Jitter

```
`include "disciplines.vams"

module osc (out);

output out; electrical out;

parameter real freq=1 from (0:inf);
parameter real Vlo=–1, Vhi=1;
parameter real tt=0.01/freq from (0:inf);
parameter real jitter=0 from [0:0.1/freq);      // period jitter

integer n, seed;
real next, dT;

analog begin
    @(initial_step) begin
        seed = 286;
        next = 0.5/freq + $abstime;
    end

    @(timer(next)) begin
        n = !n;
        dT = jitter*$rdist_normal(seed,0,1);
        next = next + 0.5/freq + 0.707*dT;
    end

    V(out) <+ transition(n ? Vhi : Vlo, 0, tt);
end
endmodule
```

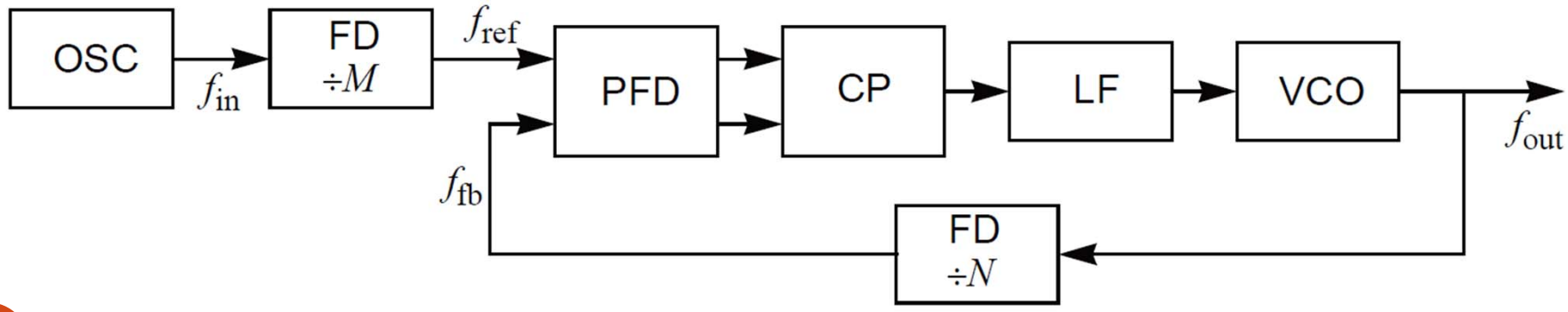
Jitters in a PLL

- Assume that the PLL has a closed-loop bandwidth of *f*L, and that $\tau_L = 1/2\pi f_L$, then for *k* such that kT << τ_L, jitter from the VCO dominates and the PLL exhibits simple accumulating jitter equal to that produced by the VCO.
- Similarly, at large *k* (low frequencies), the PLL exhibits simple accumulating jitter equal to that produced by the OSC.
- Between these two extremes, the PLL exhibits simple synchronous jitter. The amount of which depends on FDs and the PFD/CP.



Efficiency of the Models

- Conceptually, a model that includes jitter should be just as efficient as one that does not because jitter does not increase the activity of the models, it only affects the timing of particular events.
- However, if jitter causes two events that would normally occur at the same time to be displaced so that they are no longer coincident.
- For this reason, it is desirable to combine jitter sources to the degree possible.

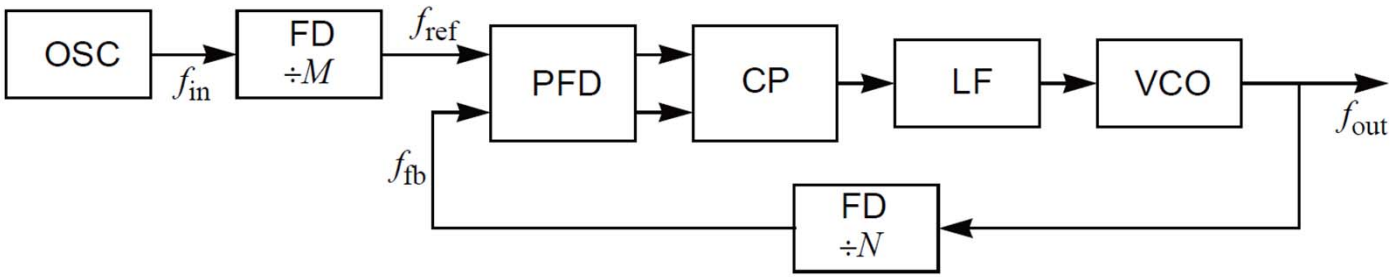


Merging OSC and FD/PFD/CP Jitters

```
`include "disciplines.vams"

module osc (out);

output out; electrical out;

parameter real freq=1 from (0:inf);
parameter real ratio=1 from (0:inf);
parameter real Vlo=–1, Vhi=1;
parameter real tt=0.01*ratio/freq from (0:inf);
parameter real accJitter=0 from [0:0.1/freq);          // period jitter
parameter real syncJitter=0 from [0:0.1*ratio/freq);   // edge-to-edge jitter

integer n, accSeed, syncSeed;
real next, dT, dt, accSD, syncSD;

analog begin
    @(initial_step) begin
        accSeed = 286;
        syncSeed = –459;
        accSD = accJitter*sqrt(ratio/2);
        syncSD = syncJitter;
        next = 0.5/freq + $abstime;
    end

    @(timer(next + dt)) begin
        n = !n;
        dT = accSD*$rdist_normal(accSeed,0,1);
        dt = syncSD*$rdist_normal(syncSeed,0,1);
        next = next + 0.5*ratio/freq + dT;
    end

    V(out) <+ transition(n ? Vhi : Vlo, 0, tt);
end
endmodule
```

Merging the VCO and FDN

- If the output of the VCO is not used to drive circuitry external to the synthesizer, if the divider exhibits simple synchronous jitter, and if the VCO exhibits simple accumulating jitter, then it is possible to include the frequency division aspect of the FD*N* as part of the VCO by simply adjusting the VCO gain and jitter.
- If the divide ratio of FD*N* is large, the simulation runs much faster because the high VCO output frequency is never gener-ated.



- Recall that the synchronous jitter of FD*M* and FD*N* has already been included as part of OSC, so the divider model incorporated into the VCO is noiseless and the jitter at the output of the noiseless divider results only from the VCO jitter.

Merging the VCO and FDN

- Since the divider outputs one pulse for every *N* pulses at its input, the variance in the output period is the sum of the variance in *N* input periods.
- Thus, the period jitter at the output, J_{FD}, is times larger than the period jitter at the input, J_{VCO}, or

$$J_{\mathrm{FD}} = \sqrt{N} J_{\mathrm{VCO}}$$

- Thus, to merge the divider into the VCO, the VCO gain must be reduced by a factor of *N,* the period jitter increased by a factor of $\sqrt{N}$, and the divider model removed.

Post Processing

- After simulation, it is necessary to refer the computed results, which are from the output of the divider, to the output of VCO, which is the true output of the PLL.

Frequency	*Jitter*	*Phase Noise*
$f_{\mathrm{VCO}} = Nf_{\mathrm{FD}}$	$J_{\mathrm{VCO}} = \frac{J_{\mathrm{FD}}}{\sqrt{N}}$	$S_{\phi_{\mathrm{VCO}}} = N^2 S_{\phi_{FD}}$

- **See** Ken Kundert **,** Predicting the Phase Noise and Jitter of PLL-Based Frequency Synthesizers, 2006

Summary

- Basic of Verilog-A
- Modeling PLL with Verilog-A